Group: C1

SAE 1.012 report

Maxime Rastelli
Eddy Francou

Introduction

The goal of the whole project is to create an automatic classification system for a list of news that would choose which category (politics, sports, culture, economy and environment-sciences) is associated with each spares. For this,

What we made

First, we manually made the dictionaries that will be used to predict the categories. Then we created a class named **`PaireChaineEntier`** in order to give each word a number, called weight. We also made the class **`UtilitairePaireChaineEntier`**. Like its name suggests, it provides some utility functions like **`indicePourChaine`** (searching the index of a word in an array of **`PaireChaineEntier`**(s) and returning -1 if it doesn't exists), **`entierPourChaine`** (searching the integer associated to the specified word, and returning 0 if it doesn't exists), **`chaineMax`** (searching the String **`chaine`** associated with the highest integer of the specified array) and **`moyenne`** (calculating the average of the integer values in **`listePaires`**).

Then, we created some methods in **`Classification.java`**. Some are reserved for testing the code, some others for displaying the vectors or the dictionaries and one that is in charge of sorting all the news of the variable **`depeches`**, for example. We also created the function **`score`** in **`Categorie.java`** which calculates the score of each news for each category. Finally, the main program uses all of what is mentioned above (and more) to give categories to wires.

The second goal is to automatically create the dictionaries we made earlier and classify the news with them. For this, we created functions and methods into **`Classement.java`**, the first is **`initDico`** which returns a vector **`PaireChaineEntier`** with all the words that appear at least once in the news. We have also made another method **`calculScores`** that will increment the score of words that appear multiple times in all the news and correspond to the right category or it will decrement them if they appear multiple times but doesn't correspond to the right category. In addition, we made the function **poidsPourScore** that associates a weight for each score. Lastly, we've made the method **`generationLexique`** that will create an automatic dictionary with **`initDico`**. Again, the main program uses these methods to automatically generate the results below.

We then made another **`calculScores`** method that uses a different approach. Instead of incrementing or decrementing the **`score`** value, it counts the total for words in the correct category, and words in the wrong category. Then the score is set to (number of words in the correct category) / (number of words in the wrong category), unless the latter is equal to 0 (that makes the division impossible). In that case, the score is simply the number of words in the correct category.

The results

This table shows the comparison between our dictionary and the automatic dictionary. Each percentage corresponds to the number of times the program's answer is correct (comparing the "guessed" category to the real category of a given wire).

		Human-made dictionary	Automatically generated dictionary	
			subtract	divide (alternate method)
Results for model wires	ENVIRONMENT AND SCIENCES	79%	86%	97%
	CULTURE	82%	79%	92%
	ECONOMY	80%	89%	88%
	POLITICS	72%	94%	95%
	SPORTS	90%	96%	97%
	AVERAGE	80.6%	88.8%	93.8%
Results for test wires	ENVIRONMENT AND SCIENCES	70%	61%	73%
	CULTURE	81%	64%	67%
	ECONOMY	69%	62%	63%
	POLITICS	68%	76%	68%
	SPORTS	84%	90%	86%
	AVERAGE	74.4%	70.6%	71.4%

As we can see, our dictionaries are more efficient than the automatic for the test wires but not for the model wires (the ones used to make the dictionaries). The category where the program gets the most correct answers is the sports category. Except for this, the average is around 60-70%.

The complexity

In score

```java
public int score(Depeche d) {
    int score = 0;
    for (int i = 0; i < d.getMots().size(); i++) {
        score += UtilitairePaireChaineEntier.entierPourChaine(lexique, d.getMots().get(i));
    }
    return score;
}
```

The function itself only consists of a read through all the words of the wire given as an argument, and there is no comparison. But it calls a method from **`UtilitairePaireChaineEntier`** from which complexity must be calculated .

For this purpose, we decided to fuse the two methods to make the analysis clearer. This is the number of executions for each barometer :

```java
public int score(Depeche d) {
    int score = 0;
    int j;
    String chaine;
    for (int i = 0; i < d.getMots().size(); i++) {
        j = 0;                                          // executed n times in all cases
        chaine = d.getMots().get(i);
        while (j < lexique.size() && !lexique.get(j).getChaine().equals(chaine)) {
            j++;                                        // in the best case, this is executed 1 time
                                                        // in the worst case, it is executed m times
        }
        if (j != lexique.size()) {
            score += lexique.get(j).getEntier();        // in all cases, this is executed 0 or 1 time
        }
    }
    return score;
}
```

With an asymptotic simplification, the method **`score`** is Ω(n), and O(n*m) : the method **`score`** has a quadratic complexity.

In calculScores

The method is mainly composed of 3 nested loops :

```
public static void calculScoresSub(ArrayList<Depeche> depeches, String categorie,
ArrayList<PaireChaineEntier> dictionnaire) {
    String currentWord;
    for (int i = 0; i < depeches.size(); i++) {
        for (int j = 0; j < depeches.get(i).getMots().size(); j++) {          // in all cases this is executed n times
            currentWord = depeches.get(i).getMots().get(j);                   // in all cases this is executed m times
            for (int k = 0; k < dictionnaire.size(); k++) {
                if (dictionnaire.get(k).getChaine().equals(currentWord)) {    // in all cases this is executed l times
                    if (depeches.get(i).getCategorie().equals(categorie)) {   // in the best case this is not executed
                                                                              // in the worst case this is executed 1 time
                        dictionnaire.get(k).setEntier(dictionnaire.get(k).getEntier() + 1);
                    } else {
                        dictionnaire.get(k).setEntier(dictionnaire.get(k).getEntier() - 1);
                    }
                }
            }
        }
    }
}
```

The best and the worst cases are the same, so the method **`calculScore`** is Ω(n*m*l), and O(n*m*l) : the complexity is polynomial.

Improvements made

We implemented a natural order to the dictionary (**`chaine`**, **`entier`**) with a new function **`compareTo()`** in **`PaireChaineEntier`**. Therefore, we can take advantage of it using dichotomous research.

We also changed the way to find an element in an array. Before, it was sequential : the program was searching the right value one by one. Now, it's searching with the dichotomous research : it splits in two the vector, check if the term we search is bigger or less than the middle of the vector and split in two the vector then repeat it until there is only one element left.

These improvements allow us to make the execution faster than before. As we can see on this table, the dichotomous version seems faster at all cases:

	Human Made dictionary	Automatically generated dictionary	
		subtract method	divide(alternative method)

Unsorted lists	63.95 ms	426.37 ms	413.27 ms
Sorted lists with dichotomous research	24.98 ms	353.68 ms	351.25 ms

Possible improvements left

It is possible to make the program even faster by optimizing the algorithms used in it. For example, we can factorize some statements. This is possible by using a variable to store a value otherwise calculated multiple times. Apart from that, it is possible in theory to make the **`calculScore`** method's complexity quadratic (O(n*m)) instead of polynomial, with the use of the builtin **`contains`** method. Unfortunately this seems to not be possible with objects, because of the use of the custom class **`PaireChaineEntier`**, which doesn't have any **`equals`** method.